Machine Learning Prediction of Obesity Levels from Behavioral and Physical Attributes

GIT HUB LINK : https://github.com/kismiskorada/Obesity-Prediction

Sole contributor: Kusuma Korada

Project Objective: This project aims to develop a Python-based machine learning model to predict obesity levels in individuals using a combination of behavioral and physical attributes. Obesity is a global health issue linked to numerous diseases, including diabetes, cardiovascular diseases, and certain types of cancer. Early detection and management can significantly reduce the risk of these health issues. This project seeks to utilize machine learning algorithms to provide an efficient, scalable, and non-invasive method to predict an individual's obesity level based on readily available data.

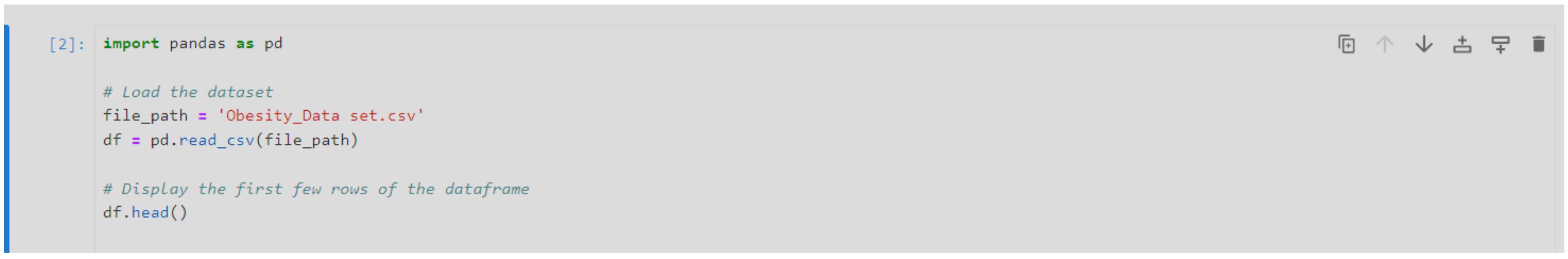
Problem Statement: The primary challenge this project intends to address is the accurate prediction of obesity levels using non-clinical, easily obtainable data. By leveraging machine learning algorithms, I aim to create a model that can assist in early detection and health management planning, potentially offering a valuable tool for healthcare professionals and individuals alike.

Dataset:
The dataset to be used is the "ObesityDataSet_raw_and_data_synthetic" which contains records of individuals along with their eating habits, physical condition, and obesity levels. The dataset is composed of both raw and synthetic data, offering a comprehensive set of features for analysis. The dataset is obtained from UCI Machine Learning Repository and the the link to this dataset is https://archive.ics.uci.edu/dataset/544/estimation+of+obesity+levels+based+on+eating+habit s+and+physical+condition

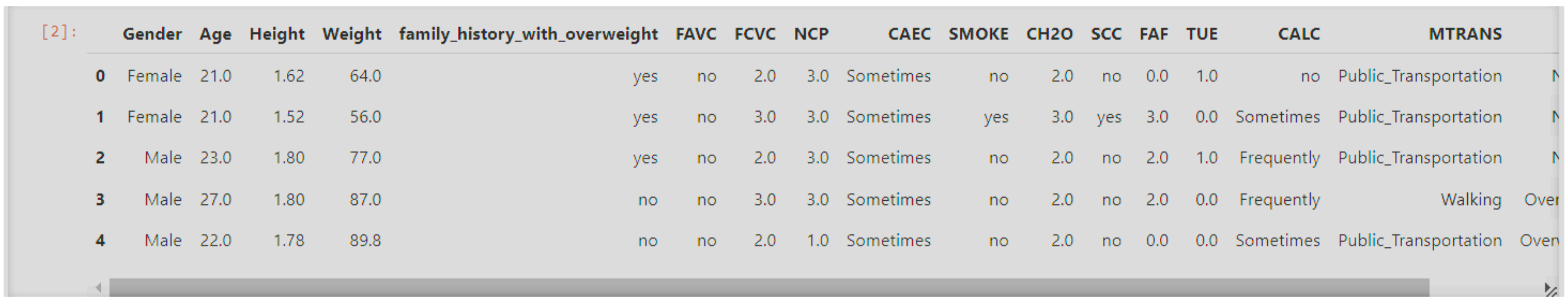
CONTENTS OF THE DATA SET:
The dataset contains the following columns: Conclusion: This project intends to harness the power of machine learning to offer insights into obesity prediction based on non-clinical factors. By doing so, it aims to contribute to the early detection and management of obesity, ultimately aiding in the prevention of related health issues.
Gender: The gender of the individual (Male/Female).
Age: The age of the individual.
Height: The height of the individual in meters.
Weight: The weight of the individual in kilograms. family_history_with_overweight: Whether there is a family history of being overweight (yes/no).
FAVC: Frequent consumption of high caloric food (yes/no).
FCVC: Frequency of consumption of vegetables (numeric scale). NCP: Number of main meals (numeric scale).
CAEC: Consumption of food between meals (Never, Sometimes, Frequently, Always).
SMOKE: Smoking status (yes/no).
CH2O: Consumption of water daily (numeric scale).
SCC: Calories consumption monitoring (yes/no).
FAF: Physical activity frequency (numeric scale).
TUE: Time using technology devices (numeric scale).
CALC: Consumption of alcohol (Never, Sometimes, Frequently, Always).
MTRANS: Mode of transportation (e.g., Public_Transportation, Walking).
NObeyesdad: Obesity level classification (e.g., Normal_Weight, Overweight_Level_I, Overweight_Level_II)

LOADING OF DATA SET:

The data set is loaded in a CSV format into a pandas DataFrame and displaying the first few rows of the DataFrame to give a glimpse of the data.



```
[2]: import pandas as pd

     # Load the dataset
     file_path = 'Obesity_Data set.csv'
     df = pd.read_csv(file_path)

     # Display the first few rows of the dataframe
     df.head()
```



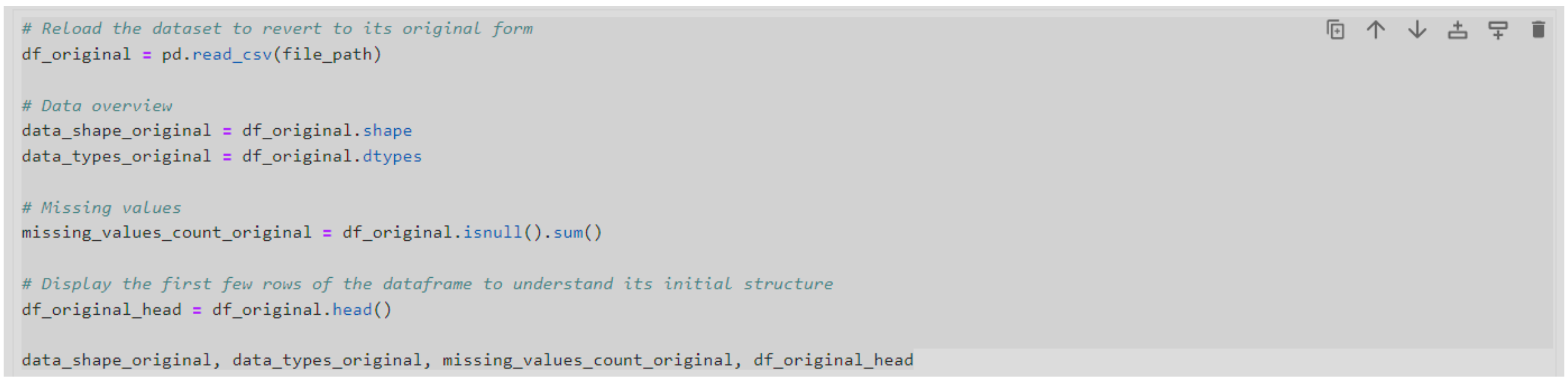
```
[2]:   Gender  Age  Height  Weight  family_history_with_overweight  FAVC  FCVC  NCP       CAEC  SMOKE  CH2O  SCC  FAF  TUE       CALC                 MTRANS
     0 Female  21.0   1.62    64.0                             yes    no   2.0  3.0  Sometimes     no   2.0   no  0.0  1.0         no  Public_Transportation     N
     1 Female  21.0   1.52    56.0                             yes    no   3.0  3.0  Sometimes    yes   3.0  yes  3.0  0.0  Sometimes  Public_Transportation     N
     2   Male  23.0   1.80    77.0                             yes    no   2.0  3.0  Sometimes     no   2.0   no  2.0  1.0  Frequently Public_Transportation     N
     3   Male  27.0   1.80    87.0                              no    no   3.0  3.0  Sometimes     no   2.0   no  2.0  0.0  Frequently               Walking  Over
     4   Male  22.0   1.78    89.8                              no    no   2.0  1.0  Sometimes     no   2.0   no  0.0  0.0  Sometimes  Public_Transportation  Overw
```

This dataset includes both behavioral and physical attributes of individuals, such as Gender, Age, Height, Weight, eating habits, and physical activity, to use for predicting obesity levels.

EXPLORATORY DATA ANALYSIS:

This Python code is reloading the original dataset from the CSV file and then providing an overview of its structure and content. Let's break down each part:
1.df_original = pd.read_csv(file_path): reloads the original dataset from the CSV file specified by file_path into a new DataFrame called df_original.
2.data_shape_original = df_original.shape: calculates the shape of the DataFrame df_original, which returns a tuple containing the number of rows and columns in the DataFrame. The result is stored in the variable data_shape_original.
3.data_types_original = df_original.dtypes: determines the data types of each column in the DataFrame df_original using the dtypes attribute. The result is stored in the variable data_types_original.
4.missing_values_count_original = df_original.isnull().sum(): calculates the count of missing values for each column in the DataFrame df_original. It uses the isnull() function to identify missing values, followed by sum() to count them. The result is stored in the variable missing_values_count_original.
5.df_original_head = df_original.head(): displays the first few rows of the DataFrame df_original to understand its initial structure. The result is stored in the variable df_original_head.
Finally, the code returns a tuple containing:

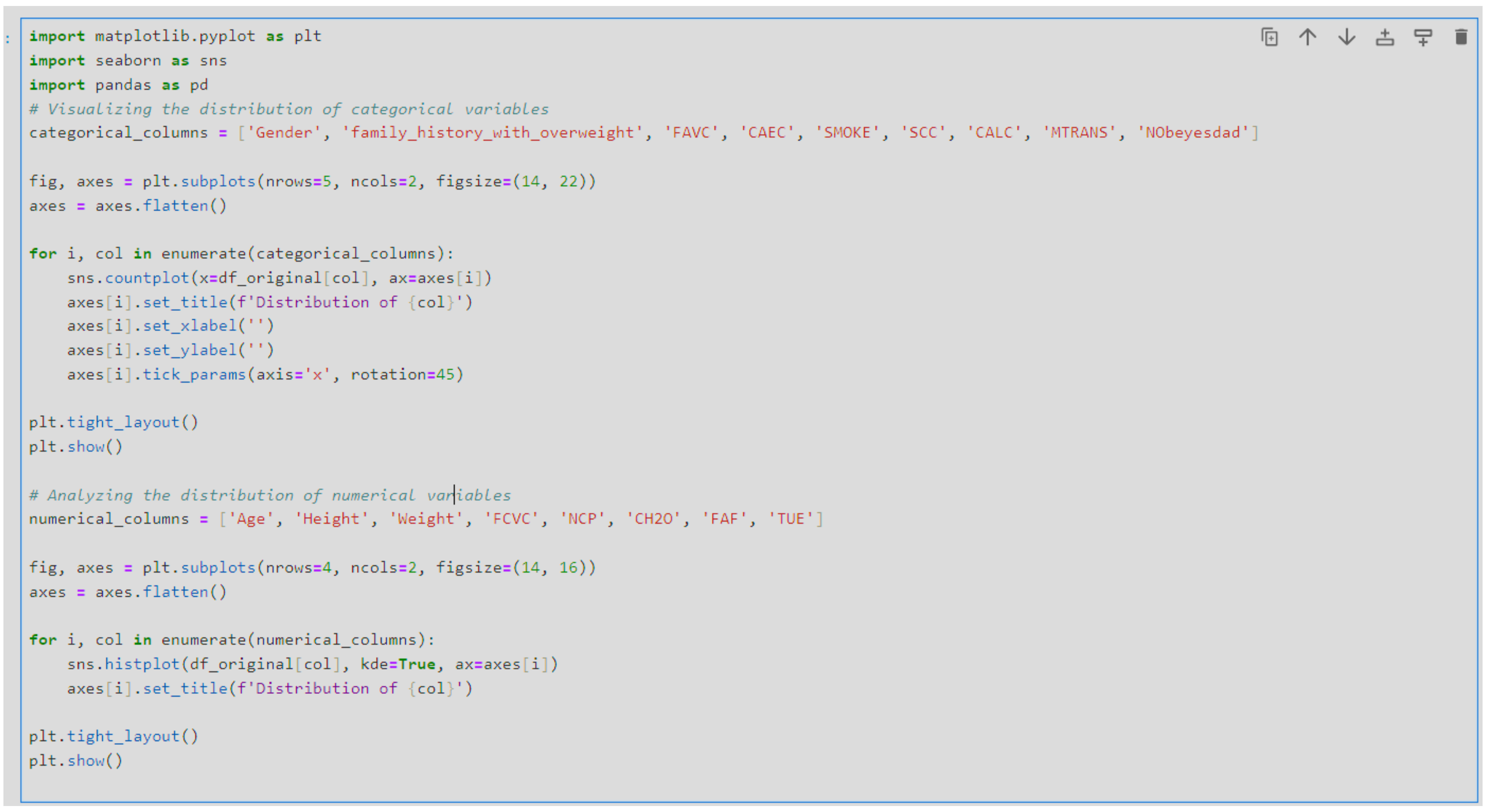
1. data_shape_original: the shape (number of rows and columns) of the original DataFrame.
2. data_types_original: the data types of each column in the original DataFrame.
3. missing_values_count_original: the count of missing values for each column in the original DataFrame.
   df_original_head: the first few rows of the original DataFrame.



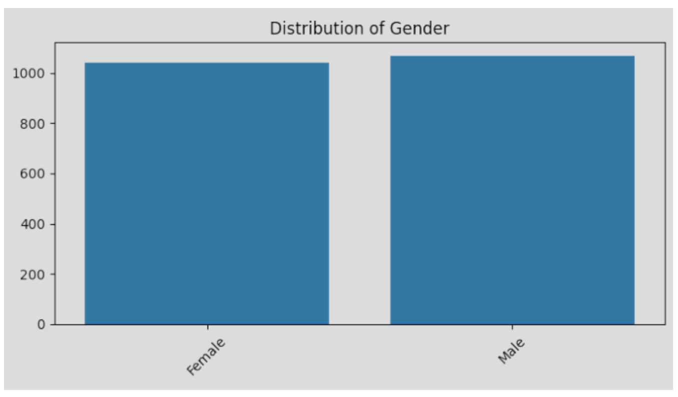
```python
# Reload the dataset to revert to its original form
df_original = pd.read_csv(file_path)

# Data overview
data_shape_original = df_original.shape
data_types_original = df_original.dtypes

# Missing values
missing_values_count_original = df_original.isnull().sum()

# Display the first few rows of the dataframe to understand its initial structure
df_original_head = df_original.head()

data_shape_original, data_types_original, missing_values_count_original, df_original_head
```

Dataset Overview Total Rows: 2111 Total Columns: 17 Data Types:The dataset contains a mix of object (categorical variables), float64 (numerical variables), and a single int64 variable. Specifically, we have categorical variables such as Gender, family_history_with_overweight, FAVC, CAEC, SMOKE, SCC, CALC, and MTRANS, along with the target variable NObeyesdad. Missing Values:There are no missing values across any of the columns in the dataset, which simplifies the preprocessing steps since we won't need to impute or remove missing data. First Few Rows:The first few rows show a variety of features that include both behavioral and physical attributes of individuals, such as Age, Height, Weight, dietary habits (FAVC, FCVC, NCP, CAEC, CH2O, CALC), lifestyle habits (SMOKE, SCC, FAF, TUE), mode of transportation (MTRANS), and the target variable NObeyesdad indicating the obesity level.

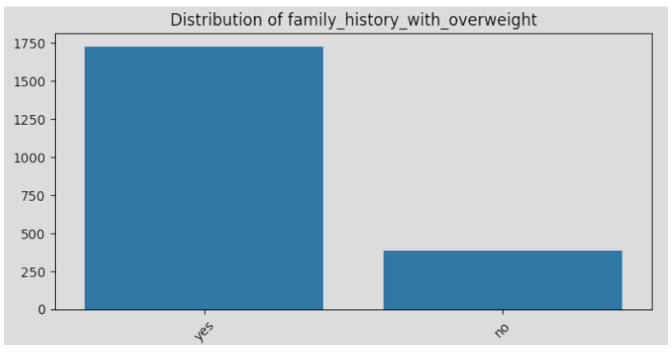
VISUALISATION OF THE DATA SET:



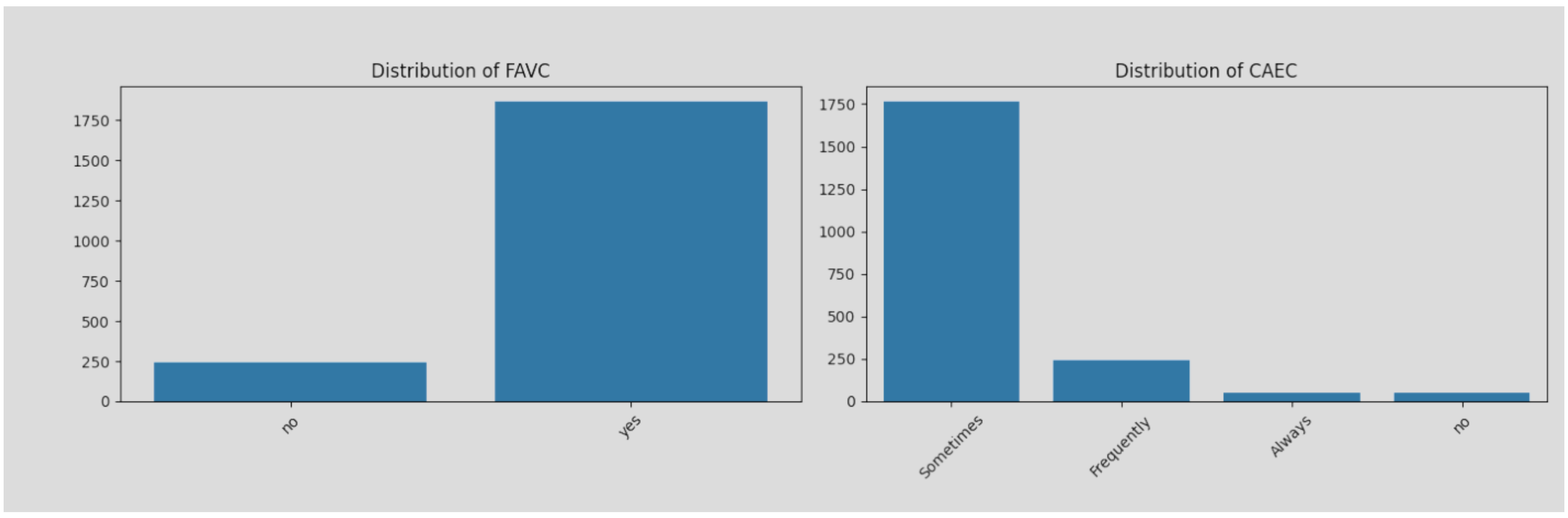
```python
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
# Visualizing the distribution of categorical variables
categorical_columns = ['Gender', 'family_history_with_overweight', 'FAVC', 'CAEC', 'SMOKE', 'SCC', 'CALC', 'MTRANS', 'NObeyesdad']

fig, axes = plt.subplots(nrows=5, ncols=2, figsize=(14, 22))
axes = axes.flatten()

for i, col in enumerate(categorical_columns):
    sns.countplot(x=df_original[col], ax=axes[i])
    axes[i].set_title(f'Distribution of {col}')
    axes[i].set_xlabel('')
    axes[i].set_ylabel('')
    axes[i].tick_params(axis='x', rotation=45)

plt.tight_layout()
plt.show()

# Analyzing the distribution of numerical variables
numerical_columns = ['Age', 'Height', 'Weight', 'FCVC', 'NCP', 'CH2O', 'FAF', 'TUE']

fig, axes = plt.subplots(nrows=4, ncols=2, figsize=(14, 16))
axes = axes.flatten()

for i, col in enumerate(numerical_columns):
    sns.histplot(df_original[col], kde=True, ax=axes[i])
    axes[i].set_title(f'Distribution of {col}')

plt.tight_layout()
plt.show()
```



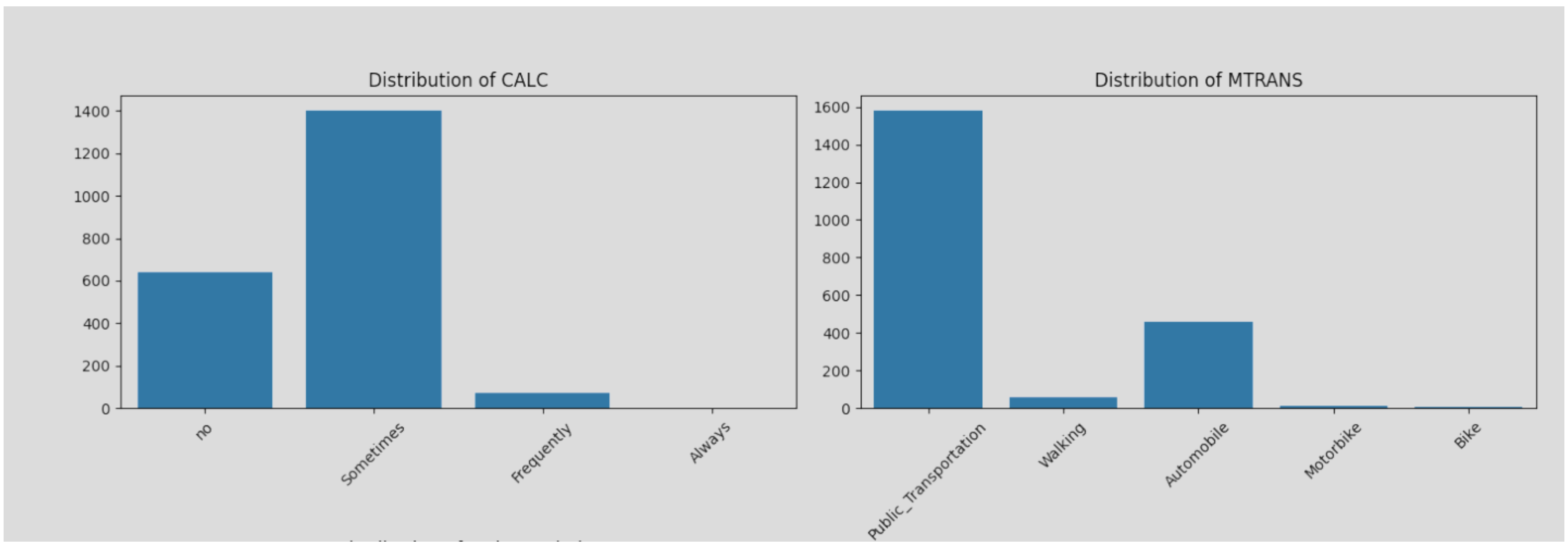
Gender: The dataset has a relatively balanced distribution of male and female participants.



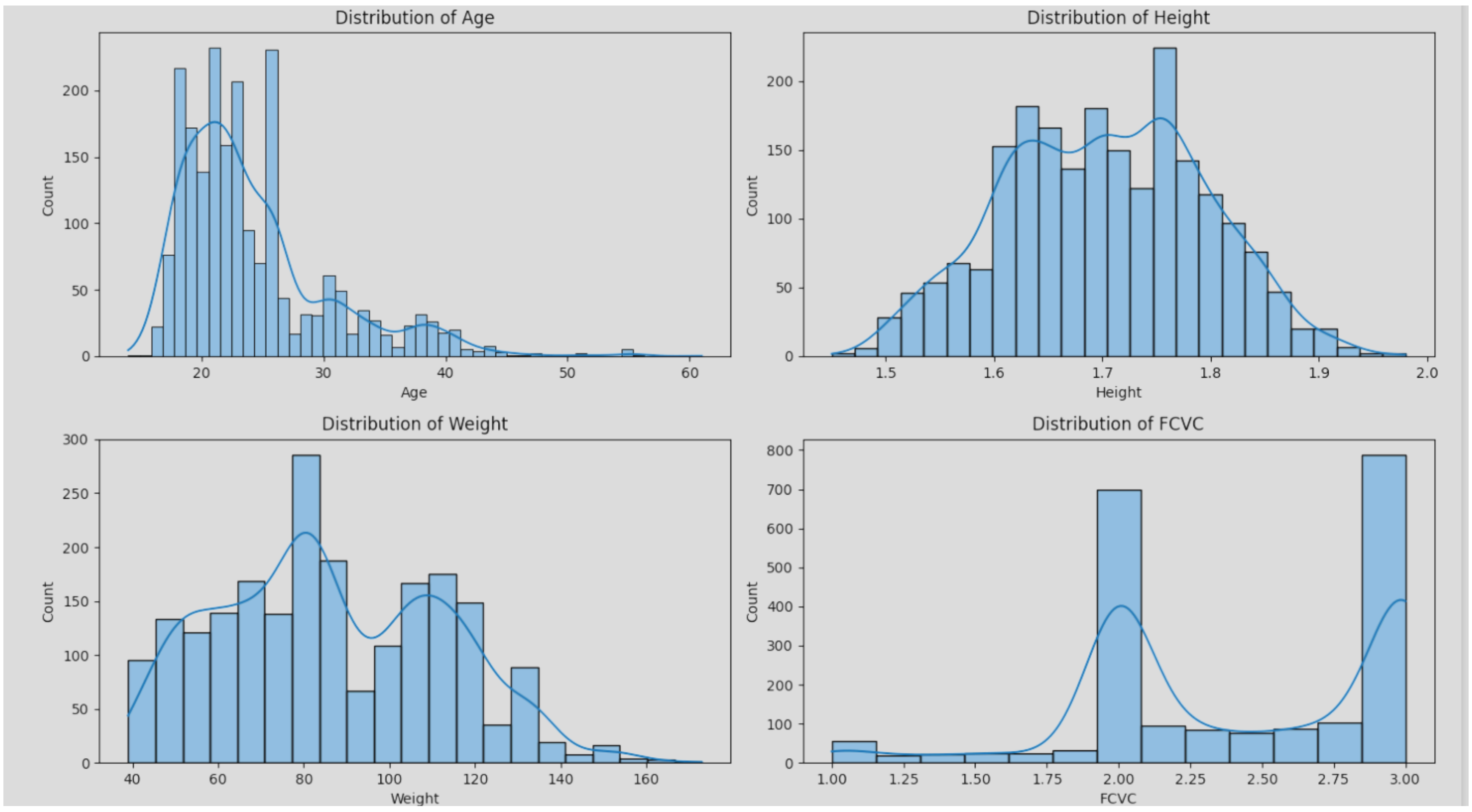
Family History with Overweight: A significant portion of the individuals has a family history of being overweight.

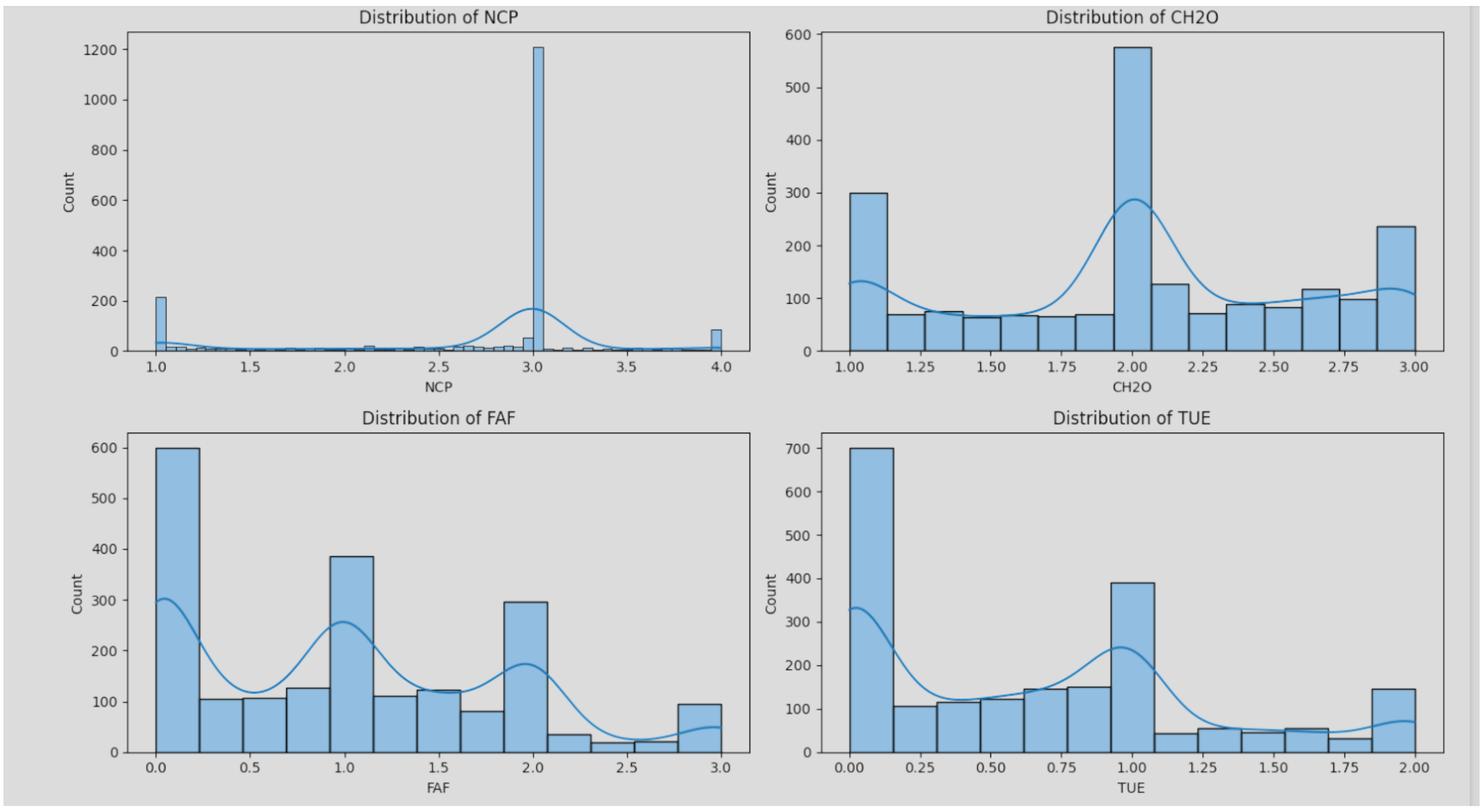


FAVC (Frequent consumption of high caloric food): Most participants frequently consume high caloric food. CAEC (Consumption of food between meals): The majority of participants sometimes or never consume food between meals.



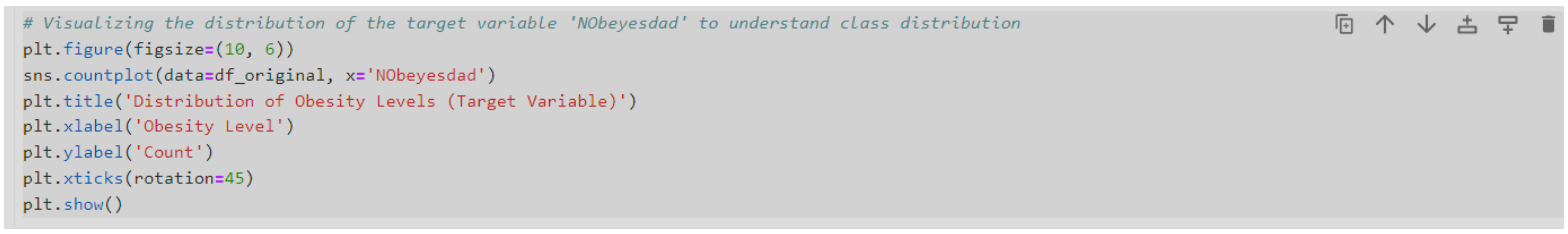
CALC (Consumption of alcohol): Alcohol consumption varies, with "Sometimes" being the most common response. MTRANS (Mode of Transportation): Public transportation is the most common mode of transportation among participants.





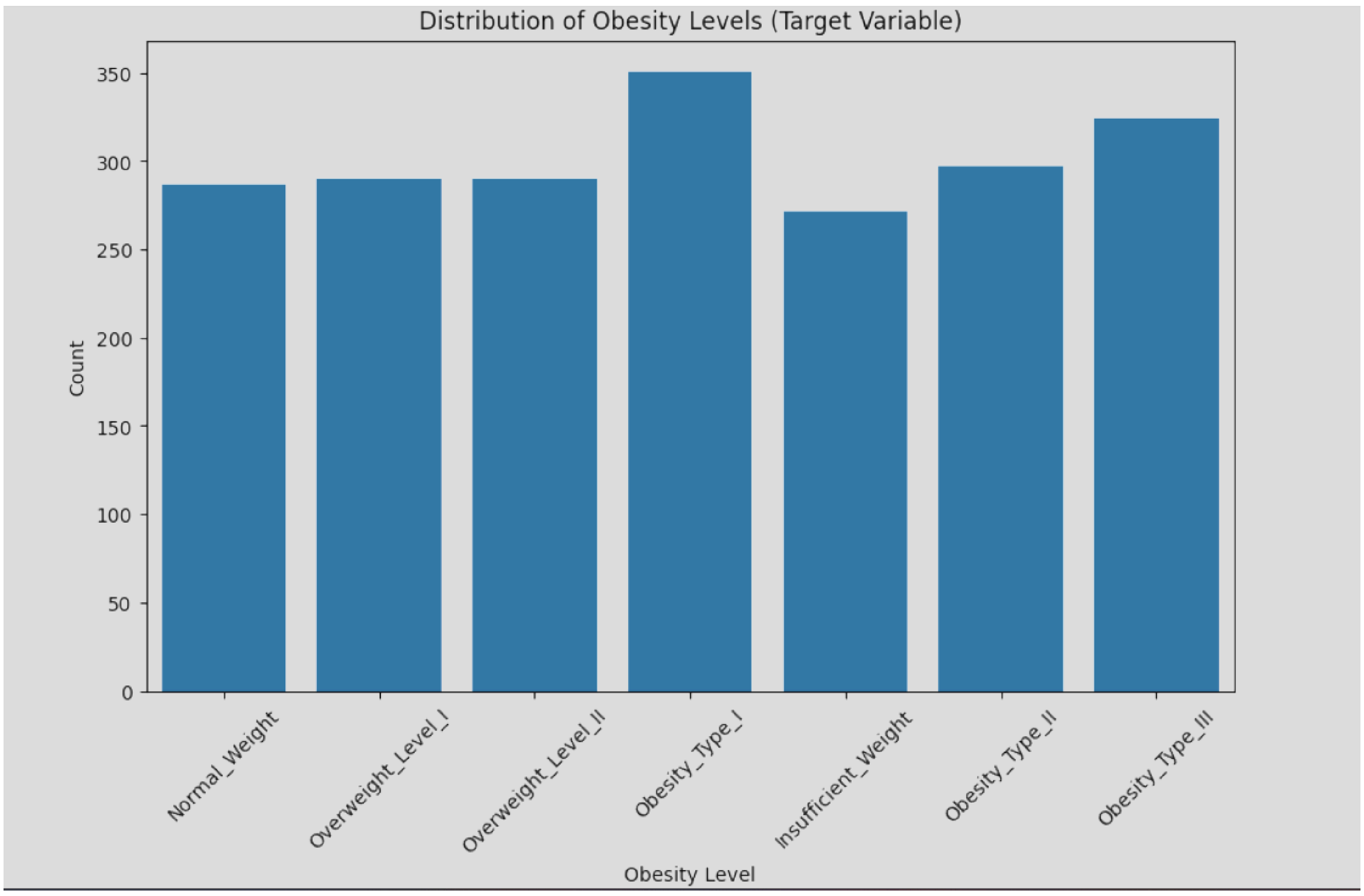
Age: The distribution is slightly right-skewed, indicating a younger population in the dataset. Height and Weight: Both show a broad range of values, with height being normally distributed and weight showing a slight right skew. FCVC (Frequency of consumption of vegetables), NCP (Number of main meals), CH2O (Consumption of water daily): These variables show varied distributions, indicating diverse eating and drinking habits. FAF (Physical activity frequency): Indicates a right-skewed distribution, suggesting that a lower frequency of physical activity is more common. TUE (Time using technology devices): Also right-skewed, indicating that most participants spend less time using technology devices.

VISUALISATION OF THE TARGET VARIABLE “NObeyedad”



```
# Visualizing the distribution of the target variable 'NObeyesdad' to understand class distribution
plt.figure(figsize=(10, 6))
sns.countplot(data=df_original, x='NObeyesdad')
plt.title('Distribution of Obesity Levels (Target Variable)')
plt.xlabel('Obesity Level')
plt.ylabel('Count')
plt.xticks(rotation=45)
plt.show()
```

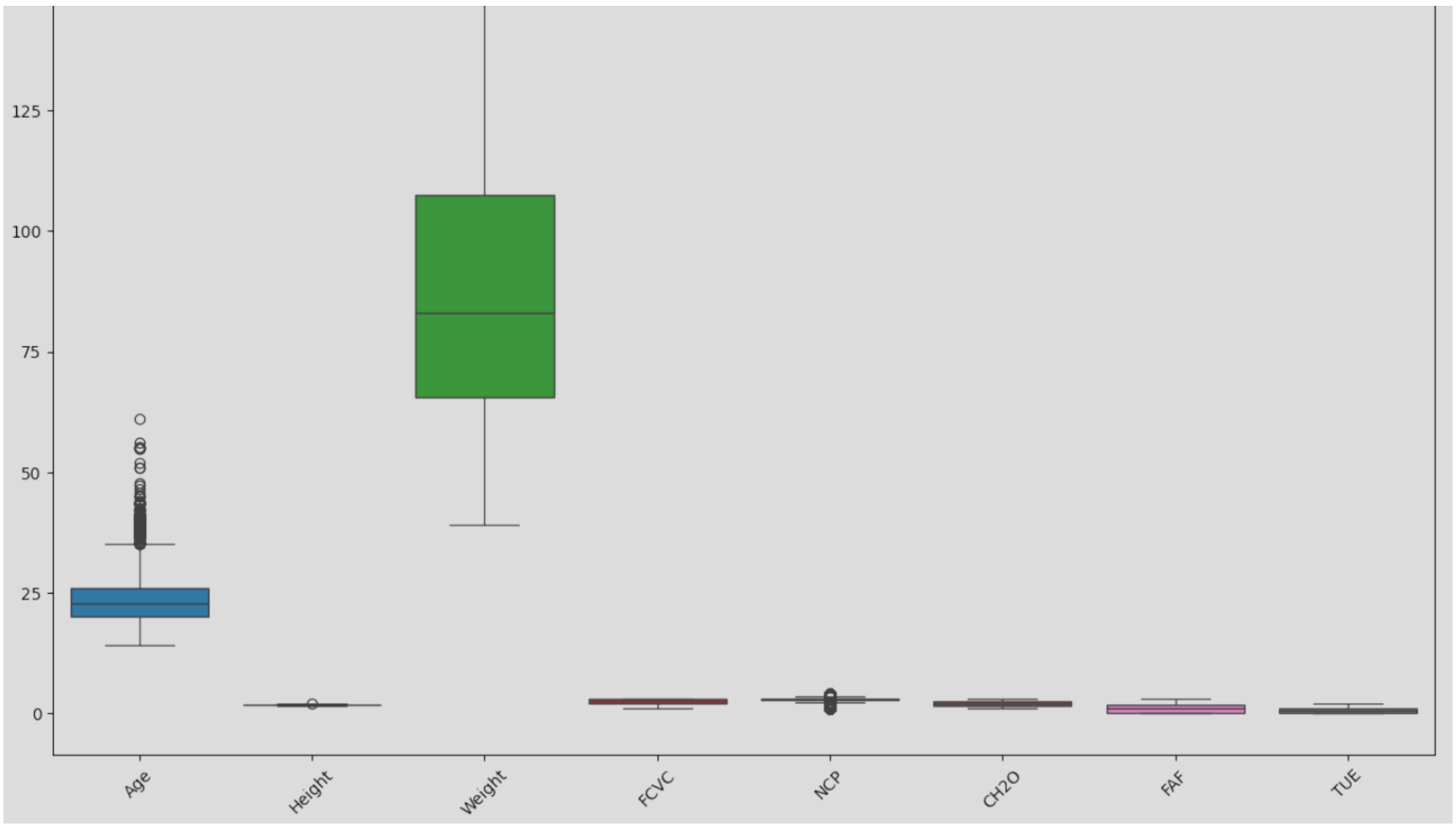
```python
import seaborn as sns
import matplotlib.pyplot as plt

# Selecting numerical columns
numerical_columns = df_original.select_dtypes(include=['float64', 'int64']).columns

# Plotting boxplots for numerical variables
plt.figure(figsize=(15, 10))
sns.boxplot(data=df_original[numerical_columns])
plt.title('Boxplot of Numerical Variables')
plt.xticks(rotation=45)
plt.show()
```



In THIS dataset, while there is some imbalance, it is not very severe, as no class is outnumbered by a very large margin.

BOX PLOTTING FOR THE DETECTION OF OUTLIERS:

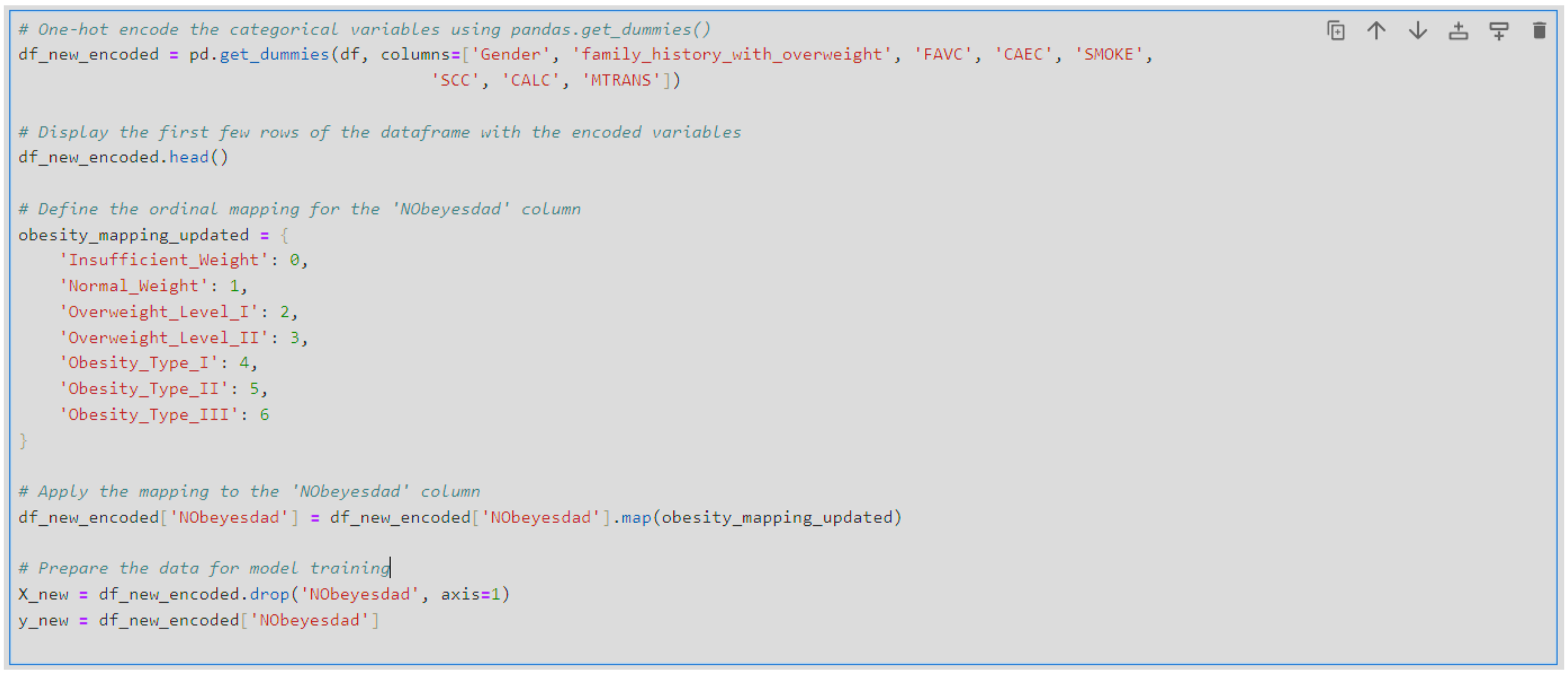
This step involves visualizing the distribution of numerical variables using boxplots.
1.Selecting numerical columns: First, we identify the numerical columns in the DataFrame df_original using the select_dtypes method with the argument include=['float64', 'int64']. This ensures that only columns with numerical data types (float64 and int64) are selected.
2.Plotting boxplots: We create a new figure with a specific size using plt.figure(figsize=(15, 10)). Then, we use sns.boxplot() to create boxplots for the numerical variables in the DataFrame df_original. We pass data=df_original[numerical_columns] to specify the data and select only the numerical columns.
3.Customizing the plot: We set the title of the plot using plt.title('Boxplot of Numerical Variables'). The plt.xticks(rotation=45) statement rotates the x-axis labels by 45 degrees to improve readability.
4.Displaying the plot: Finally, we use plt.show() to display the plot.



ENCODING:

This step involves preparing the dataset for model training by encoding categorical variables and mapping the ordinal target variable 'NObeyesdad' to numerical values.

1. One-hot encoding categorical variables: The pd.get_dummies() function is used to perform one-hot encoding on categorical variables. In this case, the columns specified in the columns parameter are one-hot encoded. Each categorical variable is split into multiple binary columns representing the presence or absence of each category.
2. Displaying the encoded dataframe: After encoding, the first few rows of the dataframe df_new_encoded are displayed using the head() function to inspect the encoded variables.
3. Defining ordinal mapping for the target variable: The obesity_mapping_updated dictionary is defined to map the ordinal classes of the 'NObeyesdad' column to numerical values. Each class is assigned a numerical value based on its order or significance.
4. Applying the mapping to the target variable: The mapping defined in obesity_mapping_updated is applied to the 'NObeyesdad' column using the map() function. This replaces the categorical labels with their corresponding numerical values.
5. Preparing data for model training: Finally, the feature matrix X_new is created by dropping the target variable column from the dataframe, and the target vector y_new is assigned the 'NObeyesdad' column, which now contains the mapped numerical values. These datasets are now ready to be used for training machine learning models.



```python
# One-hot encode the categorical variables using pandas.get_dummies()
df_new_encoded = pd.get_dummies(df, columns=['Gender', 'family_history_with_overweight', 'FAVC', 'CAEC', 'SMOKE',
                                             'SCC', 'CALC', 'MTRANS'])

# Display the first few rows of the dataframe with the encoded variables
df_new_encoded.head()

# Define the ordinal mapping for the 'NObeyesdad' column
obesity_mapping_updated = {
    'Insufficient_Weight': 0,
    'Normal_Weight': 1,
    'Overweight_Level_I': 2,
    'Overweight_Level_II': 3,
    'Obesity_Type_I': 4,
    'Obesity_Type_II': 5,
    'Obesity_Type_III': 6
}

# Apply the mapping to the 'NObeyesdad' column
df_new_encoded['NObeyesdad'] = df_new_encoded['NObeyesdad'].map(obesity_mapping_updated)

# Prepare the data for model training
X_new = df_new_encoded.drop('NObeyesdad', axis=1)
y_new = df_new_encoded['NObeyesdad']
```

TRAINING THE MODEL WITH RANDOM FOREST BAGGING CLASSIFIER:

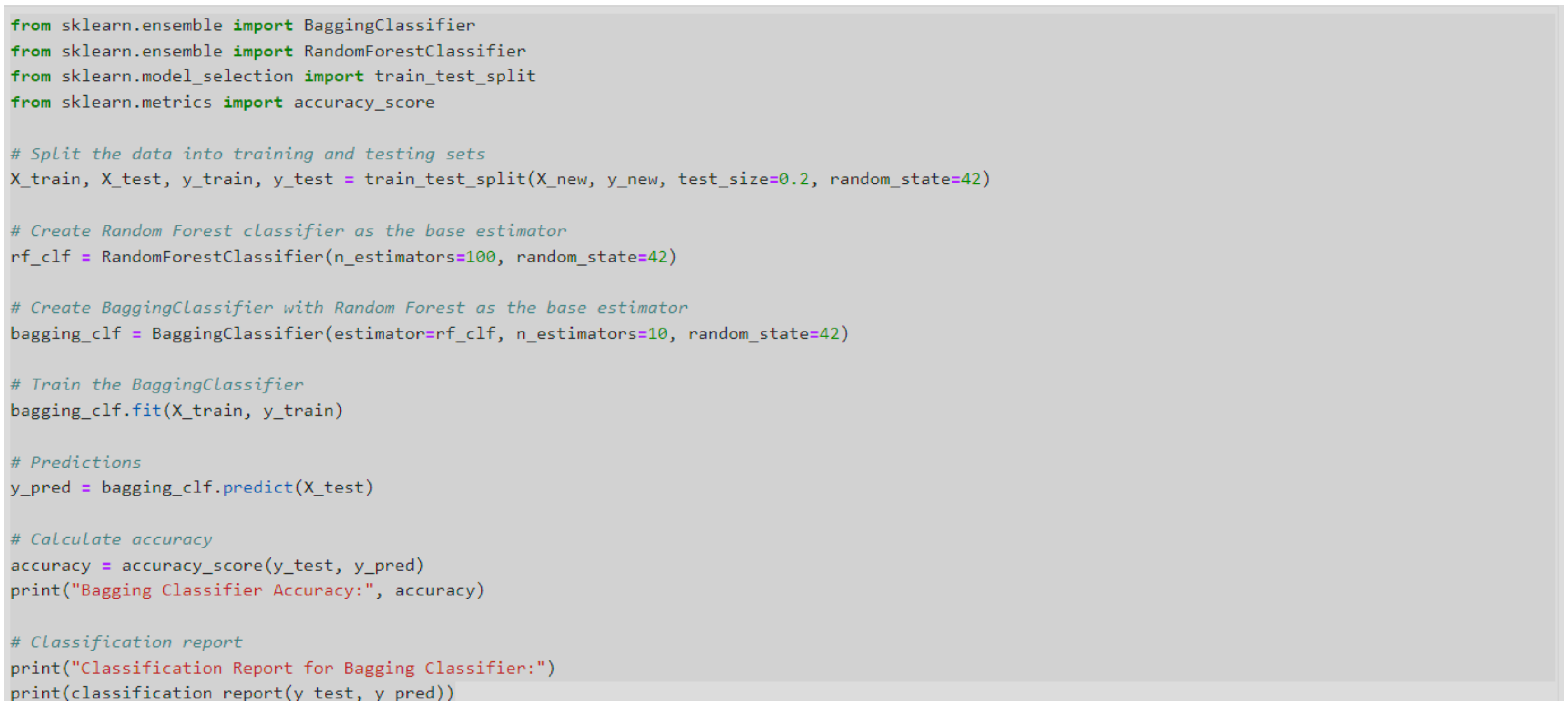
1.Data Splitting: The dataset is split into training and testing sets using the train_test_split function from scikit-learn. The feature matrix X_new and target vector y_new are split into training (X_train, y_train) and testing (X_test, y_test) sets, with a test size of 20% and a random state of 42 for reproducibility.

2.Base Estimator Creation: A Random Forest classifier (RandomForestClassifier) is created as the base estimator. It consists of 100 decision trees (n_estimators=100) and is initialized with a random state of 42 for reproducibility.

Bagging Classifier Initialization: A Bagging Classifier (BaggingClassifier) is created with the Random Forest classifier (rf_clf) as the base estimator. It consists of 10 estimators (n_estimators=10) and is also initialized with a random state of 42.

3.Model Training: The Bagging Classifier is trained on the training data (X_train, y_train) using the fit method.

4.Prediction: Predictions are made on the test data (X_test) using the trained Bagging Classifier, and the predicted labels are stored in y_pred.
5.Accuracy Calculation: The accuracy of the Bagging Classifier model is calculated by comparing the predicted labels (y_pred) with the true labels (y_test) using the accuracy_score function from scikit-learn.
6.Classification Report: A classification report is printed to provide a detailed evaluation of the model's performance, including precision, recall, F1-score, and support for each class. The classification_report function from scikit-learn is used for this purpose.
HERE SCALING IS NOT NECESSARY FOR THE RANDOM FOREST BAGGING CLASSIFIER

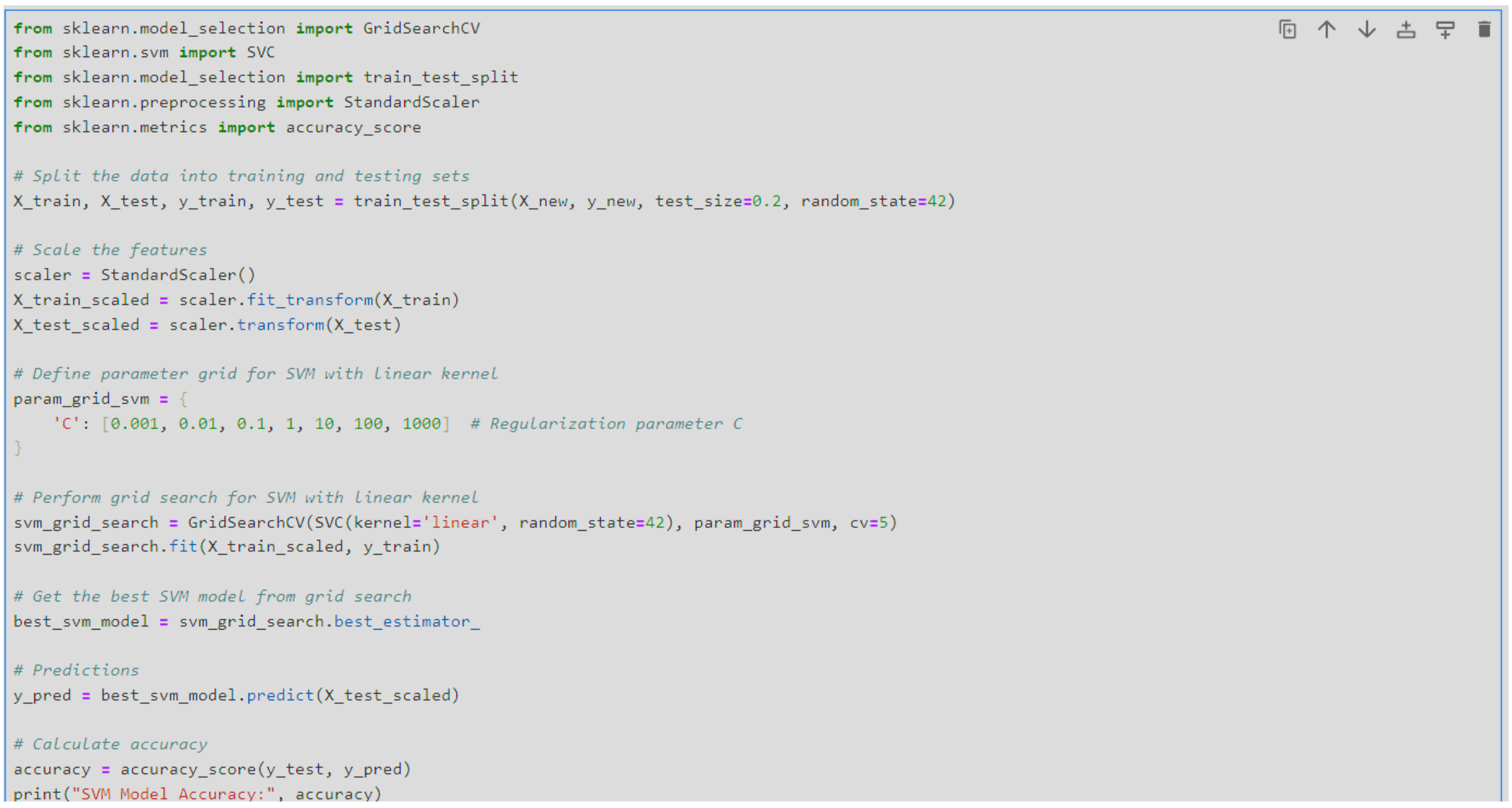


```
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_new, y_new, test_size=0.2, random_state=42)

# Create Random Forest classifier as the base estimator
rf_clf = RandomForestClassifier(n_estimators=100, random_state=42)

# Create BaggingClassifier with Random Forest as the base estimator
bagging_clf = BaggingClassifier(estimator=rf_clf, n_estimators=10, random_state=42)

# Train the BaggingClassifier
bagging_clf.fit(X_train, y_train)

# Predictions
y_pred = bagging_clf.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Bagging Classifier Accuracy:", accuracy)

# Classification report
print("Classification Report for Bagging Classifier:")
print(classification_report(y_test, y_pred))
```

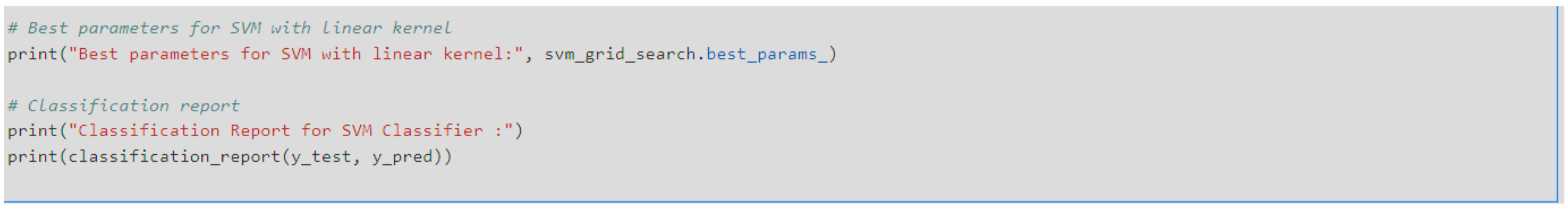
```
Bagging Classifier Accuracy: 0.9432624113475178
Classification Report for Bagging Classifier:
           precision    recall  f1-score   support

        0       0.98      0.96      0.97        56
        1       0.84      0.94      0.89        62
        2       0.91      0.86      0.88        56
        3       0.92      0.94      0.93        50
        4       0.99      0.92      0.95        78
        5       0.97      0.98      0.97        58
        6       1.00      1.00      1.00        63

   accuracy                         0.94       423
  macro avg       0.94      0.94      0.94       423
weighted avg       0.95      0.94      0.94       423
```

The classification report provides a detailed summary of the Bagging Classifier's performance on the test data:
Precision: Precision measures the proportion of true positive predictions among all positive predictions. In this report:
Precision for class 0 is 98%, indicating that 98% of the instances predicted as class 0 are actually class 0.
Precision for class 1 is 84%, meaning that 84% of the instances predicted as class 1 are actually class 1.
Precision for class 2 is 91%, indicating that 91% of the instances predicted as class 2 are actually class 2.
Precision for class 3 is 92%, meaning that 92% of the instances predicted as class 3 are actually class 3.
Precision for class 4 is 99%, indicating that 99% of the instances predicted as class 4 are actually class 4.
Precision for class 5 is 97%, meaning that 97% of the instances predicted as class 5 are actually class 5.
Precision for class 6 is 100%, indicating that all instances predicted as class 6 are actually class 6.
Recall: Recall measures the proportion of true positive predictions that were correctly identified. In this report:
Recall for class 0 is 96%, meaning that 96% of the true class 0 instances were correctly identified.
Recall for class 1 is 94%, indicating that 94% of the true class 1 instances were correctly identified.
Recall for class 2 is 86%, indicating that 86% of the true class 2 instances were correctly identified.
Recall for class 3 is 94%, meaning that 94% of the true class 3 instances were correctly identified.
Recall for class 4 is 92%, indicating that 92% of the true class 4 instances were correctly identified.
Recall for class 5 is 98%, meaning that 98% of the true class 5 instances were correctly identified.
Recall for class 6 is 100%, indicating that all true class 6 instances were correctly identified.
F1-score: The F1-score is the harmonic mean of precision and recall and provides a balance between the two metrics. In this report:
The F1-score ranges from 88% to 100%, indicating the balance between precision and recall for each class.
Support: Support represents the number of actual occurrences of each class in the test data.
Macro Avg: The macro average calculates metrics for each class independently and then takes the average. It provides equal weight to each class, regardless of class imbalance.
Weighted Avg: The weighted average calculates metrics for each class independently and then takes the weighted average based on the number of true instances for each class. It gives higher weight to classes with more instances, which is useful for imbalanced datasets.
Overall, the Bagging Classifier demonstrates good performance with an accuracy of 94.3%, as well as high precision, recall, and F1-scores for most classes.

TRAINING THE MODEL WITH SVM:
SCALING IS NECESSARY



```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_new, y_new, test_size=0.2, random_state=42)

# Scale the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Define parameter grid for SVM with linear kernel
param_grid_svm = {
    'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000]  # Regularization parameter C
}

# Perform grid search for SVM with linear kernel
svm_grid_search = GridSearchCV(SVC(kernel='linear', random_state=42), param_grid_svm, cv=5)
svm_grid_search.fit(X_train_scaled, y_train)

# Get the best SVM model from grid search
best_svm_model = svm_grid_search.best_estimator_

# Predictions
y_pred = best_svm_model.predict(X_test_scaled)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("SVM Model Accuracy:", accuracy)
```



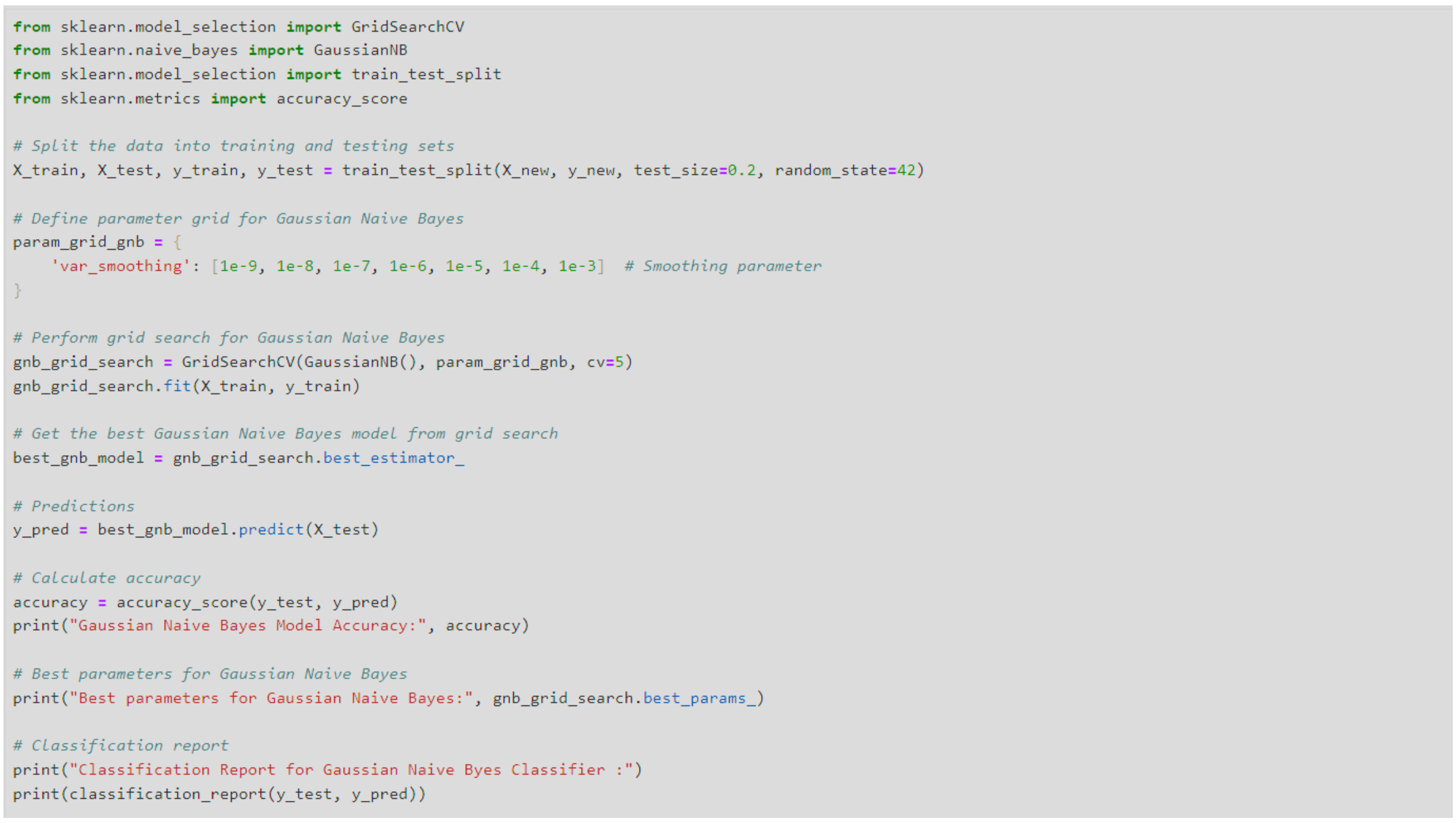
```
# Best parameters for SVM with linear kernel
print("Best parameters for SVM with linear kernel:", svm_grid_search.best_params_)

# Classification report
print("Classification Report for SVM Classifier :")
print(classification_report(y_test, y_pred))
```

The classification report provides a detailed summary of the SVM Classifier's performance on the test data:

- Precision: Precision measures the proportion of true positive predictions among all positive predictions. In this report:
  - Precision for class 0 is 98%, indicating that 98% of the instances predicted as class 0 are actually class 0.
  - Precision for class 1 is 95%, meaning that 95% of the instances predicted as class 1 are actually class 1.
  - Precision for class 2 is 88%, indicating that 88% of the instances predicted as class 2 are actually class 2.
  - Precision for class 3 is 98%, meaning that 98% of the instances predicted as class 3 are actually class 3.
  - Precision for class 4 is 97%, indicating that 97% of the instances predicted as class 4 are actually class 4.
  - Precision for class 5 is 100%, meaning that 100% of the instances predicted as class 5 are actually class 5.
  - Precision for class 6 is 100%, indicating that 100% of the instances predicted as class 6 are actually class 6.
- Recall: Recall measures the proportion of true positive predictions that were correctly identified. In this report:
  - Recall for class 0 is 98%, meaning that 98% of the true class 0 instances were correctly identified.
  - Recall for class 1 is 89%, indicating that 89% of the true class 1 instances were correctly identified.
  - Recall for class 2 is 95%, indicating that 95% of the true class 2 instances were correctly identified.
  - Recall for class 3 is 94%, meaning that 94% of the true class 3 instances were correctly identified.
  - Recall for class 4 is 100%, indicating that 100% of the true class 4 instances were correctly identified.
  - Recall for class 5 is 100%, meaning that 100% of the true class 5 instances were correctly identified.
  - Recall for class 6 is 100%, indicating that all true class 6 instances were correctly identified.
- F1-score: The F1-score is the harmonic mean of precision and recall and provides a balance between the two metrics. In this report:
  The F1-score ranges from 91% to 100%, indicating the balance between precision and recall for each class.
- Support: Support represents the number of actual occurrences of each class in the test data.
- Macro Avg: The macro average calculates metrics for each class independently and then takes the average. It provides equal weight to each class, regardless of class imbalance.
- Weighted Avg: The weighted average calculates metrics for each class independently and then takes the weighted average based on the number of true instances for each class. It gives higher weight to classes with more instances, which is useful for imbalanced datasets.

Overall, the SVM Classifier demonstrates excellent performance with an accuracy of 96.7%, as well as high precision, recall, and F1-scores for most classes.

TRAINING THE DATA SET WITH GUASSIAN NAIVE BYES:



```
from sklearn.model_selection import GridSearchCV
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_new, y_new, test_size=0.2, random_state=42)

# Define parameter grid for Gaussian Naive Bayes
param_grid_gnb = {
    'var_smoothing': [1e-9, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3]  # Smoothing parameter
}

# Perform grid search for Gaussian Naive Bayes
gnb_grid_search = GridSearchCV(GaussianNB(), param_grid_gnb, cv=5)
gnb_grid_search.fit(X_train, y_train)

# Get the best Gaussian Naive Bayes model from grid search
best_gnb_model = gnb_grid_search.best_estimator_

# Predictions
y_pred = best_gnb_model.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Gaussian Naive Bayes Model Accuracy:", accuracy)

# Best parameters for Gaussian Naive Bayes
print("Best parameters for Gaussian Naive Bayes:", gnb_grid_search.best_params_)

# Classification report
print("Classification Report for Gaussian Naive Byes Classifier :")
print(classification_report(y_test, y_pred))
```

Split Data: The dataset is split into training and testing sets using train_test_split.
Define Parameter Grid: A parameter grid for Gaussian Naive Bayes is defined. In this case, the grid includes different values for the var_smoothing parameter, which is a smoothing parameter.
Grid Search: Grid search is performed using GridSearchCV. It takes the Gaussian Naive Bayes classifier, the parameter grid, and the number of folds for cross-validation (cv) as inputs. The grid search identifies the best combination of parameters based on cross-validated performance on the training data.
Fit Grid Search: The grid search is fitted to the training data using the fit method.
Get Best Model: After grid search, the best Gaussian Naive Bayes model is obtained using the best_estimator_ attribute of the grid search object.
Predictions: The best model is used to make predictions on the test data.
Calculate Accuracy: The accuracy of the model is calculated by comparing the predicted labels with the true labels from the test set.
Print Results: The accuracy of the Gaussian Naive Bayes model, the best parameters found during grid search, and the classification report are printed to evaluate the model's performance.

```
Gaussian Naive Bayes Model Accuracy: 0.6453900709219859
Best parameters for Gaussian Naive Bayes: {'var_smoothing': 0.001}
Classification Report for Gaussian Naive Byes Classifier :
           precision    recall  f1-score   support

        0       0.66      0.91      0.77        56
        1       0.59      0.27      0.37        62
        2       0.49      0.45      0.47        56
        3       0.46      0.62      0.53        50
        4       0.70      0.58      0.63        78
        5       0.82      0.71      0.76        58
        6       0.75      1.00      0.86        63

   accuracy                         0.65       423
  macro avg       0.64      0.65      0.63       423
weighted avg       0.65      0.65      0.63       423
```

Gaussian Naive Bayes Model Accuracy: The accuracy of the Gaussian Naive Bayes model on the test set is approximately 64.54%. This indicates that about 64.54% of the predictions made by the model were correct.
Best Parameters for Gaussian Naive Bayes: The best parameter found during the grid search is var_smoothing: 0.001. This parameter controls the amount of smoothing applied to the data and is used to handle numerical stability issues.
Classification Report: This report provides various metrics for each class in the target variable, including precision, recall, and F1-score. Here's what each metric represents:

- Precision: The ratio of correctly predicted positive observations to the total predicted positives. In this context, it represents the accuracy of the classifier in identifying instances of each class.
- Recall: The ratio of correctly predicted positive observations to the total actual positives. In this context, it represents the ability of the classifier to capture all instances of each class.
- F1-score: The harmonic mean of precision and recall. It provides a balance between precision and recall.
- Support: The number of actual occurrences of each class in the test set.

Interpretation:
Looking at the classification report, we can observe that:

- Class 0 (Insufficient Weight), Class 5 (Obesity Type II), and Class 6 (Obesity Type III) have relatively higher precision and recall, indicating good performance in identifying these classes.
- Class 1 (Normal Weight) and Class 4 (Overweight Level II) have lower precision and recall compared to other classes, suggesting that the classifier struggles more with these classes.
- Overall, the weighted average F1-score is approximately 63%, indicating the overall performance of the classifier across all classes.

In summary, while the Gaussian Naive Bayes classifier achieves reasonable accuracy, it may struggle with certain classes, as indicated by lower precision and recall values for some classes. Further analysis and potentially exploring different classifiers or feature engineering techniques may help improve the model's performance, particularly for the classes with lower precision and recall.

CONCLUSION:

Based on these results, the SVM Classifier performs the best among the three models, with the highest accuracy and macro avg F1-score.

The Bagging Classifier with Random Forest also shows good performance but slightly lower than the SVM Classifier.
The Gaussian Naive Bayes Classifier performs the worst among the three models, with significantly lower accuracy and macro avg F1-score.
Therefore, if we consider accuracy and overall performance across different classes, the SVM Classifier appears to be the better choice for obesity prediction in this scenario.